Programming Model to Develop Supercomputer Combinatorial Solvers

Ghaith Tarawneh,* Andrey Mokhov,* Matthew Naylor,† Alex Rast,‡ Simon W. Moore,†
David B. Thomas,§ Alex Yakovlev,* Andrew Brown‡
* School of Electrical and Electronic Engineering, Newcastle University, UK
† Computer Laboratory, University of Cambridge, UK
‡ Electronics and Computer Science, University of Southampton, UK
§ Electrical and Electronic Engineering, Imperial College London, UK

***Abstract*—Novel architectures for massively parallel machines offer better scalability and the prospect of achieving linear speedup for sizable problems in many domains. The development of suitable programming models and accompanying software tools for these architectures remains one of the biggest challenges towards exploiting their full potential. We present a multi-layer software abstraction model to develop combinatorial solvers on massively-parallel machines with regular topologies. The model enables different challenges in the design and optimization of combinatorial solvers to be tackled independently (separation of concerns) while permitting problem-specific tuning and cross-layer optimization. In specific, the model decouples the issues of inter-node communication, node-level scheduling, problem mapping, mesh-level load balancing and expressing problem logic. We present an implementation of the model and use it to profile a Boolean satisfiability solver on simulated massively-parallel machines with different scales and topologies.**

I. Introduction

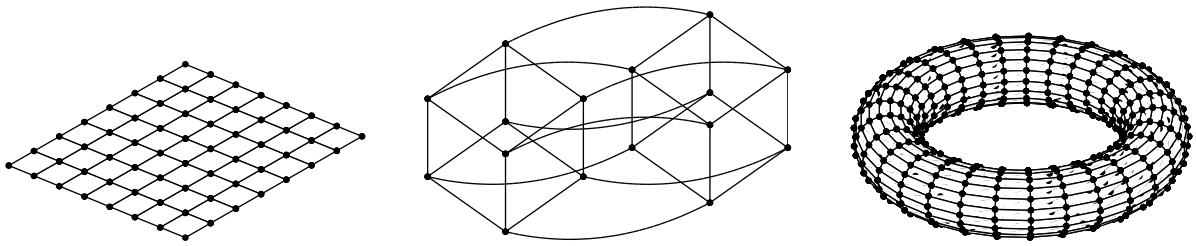
Modern massively parallel machines can now support millions of cores [1]. On this scale, full connectivity, global synchronization and complete ordering become untenable, so novel architectures emerged in which these ideals were abandoned, e.g. SpiNNaker [2]. In these architectures, cores are connected through a mesh network that is embedded in an n-dimensional space (e.g. a torus or a hypercube). Cores perform computations by exchanging messages with a limited number of other cores, usually their immediate neighbours. There is no global state and computations may proceed independently in different regions of the mesh (partial ordering). This model lends itself naturally to many problems in scientific computing where the problem domain (usually physical space) can be partitioned and mapped to the mesh of cores [3]. We refer to massively parallel machines that adopt this architecture as "hyperspace computers" (Figure 1).

Hyperspace computers offer better scalability for computational problems that can be decomposed and solved without having to exchange data between *all* sub-problems (i.e. without global communication). An important class that falls in this category is *combinatorial optimization*: problems that involve minimizing an objective function over a finite space of possible solutions. Parallelizing combinatorial solvers on conventional architectures is generally difficult because of load balancing, scheduling and communication costs [4]. These costs become increasingly prohibitive as more cores are added and therefore impose an upper bound on the speedup that can be achieved. Hyperspace computers are free from these limitations since problems can be decomposed, distributed and solved across the mesh without global communication. Cores can therefore be added without contributing to memory contention or imposing other performance limits on existing cores.

Despite these favourable scaling characteristics, developing combinatorial solvers for hyperspace computers is challenging. First, end users are often limited to using a less familiar programming model (e.g. message passing) so existing algorithms in imperative form or other styles must be re-formulated from the bottom up. Second, the programming model is often directly based on the execution model of the architecture. Users must therefore be intimately familiar with the underlying hardware and its capabilities and are likely to produce architecture-specific solutions with poor portability. Third, apart from expressing problem logic, user programs have to address other issues including problem mapping and load balancing, resulting in a poor separation of concerns.

This paper presents a software framework to tackle the above challenges. The contributions of this work are:

(1) We propose a multi-layer software abstraction model to develop combinatorial solvers for hyperspace computers. The model decouples problems that are typically addressed collectively when developing software for hyperspace computers (e.g. problem logic, mapping and load balancing) so that they can be addressed independently (separation of concerns).
(2) We present a prototype implementation of the model to showcase layer organization and separation of concerns, focusing on two issues that are particularly relevant to hyperspace computers: problem mapping and programming model conversion.
(3) We discuss an implementation of a Boolean satisfiability (SAT) solver, based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, as a use case of the model.
(4) We present empirical results obtained by running the solver from (3) on a simulator backend. We compare solver scalability for different hyperspace topologies and problem mapping algorithms.



(A) (B) (C)

Figure 1. Hyperspace computers: **(A)** transputer array in a grid configuration, **(B)** four-dimensional hypercube (e.g. NCUBE [5]) and **(C)** a modern hyperspace computer arranged as a torus (e.g. SpiNNaker [2]). All consist of relatively low-performance processors communicating via message passing.

II. Background

This section reviews the history and development of hyperspace computers. Our aim here is to present the key features of hyperspace architectures and establish a baseline for discussing their programming models.

A. Hyperspace Computers

Parallel architectures based on low-latency connections between relatively low-performance processing units date back to transputers nearly three decades ago [6]. Transputers were a family of microprocessors intended for parallel programming (Figure 1A). Each transputer unit had serial point-to-point communication links, allowing it to connect to other transputers, host computers and input/output devices. Systems based on transputers had the appeal of modularity and extensibility as nodes (transputers) could be arranged in different configurations and existing systems upgraded by adding more nodes. This provided a clear route to scalable performance and a favourable alternative to making individual microprocessors faster. Transputers were co-developed with Occam [7], a concurrent programming language based on Communicating Sequential Processes (CSP) [8] and featuring built-in operators for channel communication. Occam inherited the rigorous foundation of CSP [9], [10] and greatly simplified the development of software for transputer arrays [11].

The regular topology of hyperspace architectures was popularized by hypercube computers in the same era [12]. Hypercubes were n-dimensional binary cubes containing $N = 2^n$ processors with n neighbours each (Figure 1B). Nodes were typically assigned n bit addresses where each bit denotes position along one dimension and any two nodes whose addresses differ by a single bit were adjacent. Similar to transputers, hypercube computers emphasized having a plenitude of processors over individual processor performance. Their distinguishing feature, however, was the hypercube topology. Even though the topology was first adopted because of its efficient physical layout [13], it also offered attractive properties for software development. First, it is *node-symmetric*; all nodes have symmetric perspectives to other nodes in the system and there are no special cases. Second, communication latency and number of links both scale efficiently with the number of nodes: for 2^n nodes, there are $nN/2$ links and any two nodes are at most n links apart. Third, hypercubes can embed other topologies including trees and lower-dimensional meshes efficiently [14]–[16]. These properties contributed to the popularity of hypercubes and spawned many applications in scientific computing and combinatorial optimization [17], [18].

More recently, the high integration densities of modern VLSI processes made it possible to create massively parallel machines with thousands of times more cores than was possible during the transputer and hypercube era. This motivated the creation of SpiNNaker, a (hyperspace) supercomputer intended primarily for spiking neural network simulations (although also capable of addressing a wider range of scientific computing problems [19]). Each SpiNNaker chip contains 18 ARM9 cores and a Network-on-Chip to route packets between cores and across inter-chip links [2]. The core mesh is arranged as a torus (Figure 1C) but the underlying communication infrastructure permits arbitrary topologies to be virtualised efficiently. At the time of writing the machine consists of 500,000 cores but once complete it will house more than a million cores.

B. Programming Models

Most hyperspace computers have been co-developed with specialized programming models and software tools that reflect their architectures. The Occam language, for example, is specifically tailored to transputers (transputers were in fact developed as specialized processors to execute Occam [20]). The direct mapping between the two can be observed clearly in many of Occam's features. For example, Occam has blocking channel control operators (**!** and **?**) that force processes to synchronize when transferring data, corresponding to the synchronous links of transputers.[1] Conversely, some architectural features of transputers are adapted for Occam. Occam programs, for example, are composed of many concurrent processes with independent scopes so transputer general purpose registers were eliminated in favour of faster context switching [22]. Transputers and Occam are not the only architecture and programming model that correspond in this fashion. Hypercube computer software is

[1] Compare these operators with more flexible communication schemes such as the rendezvous mechanism in Ada where the receiver can pre-process incoming data, selectively block and return data to the sender [21].

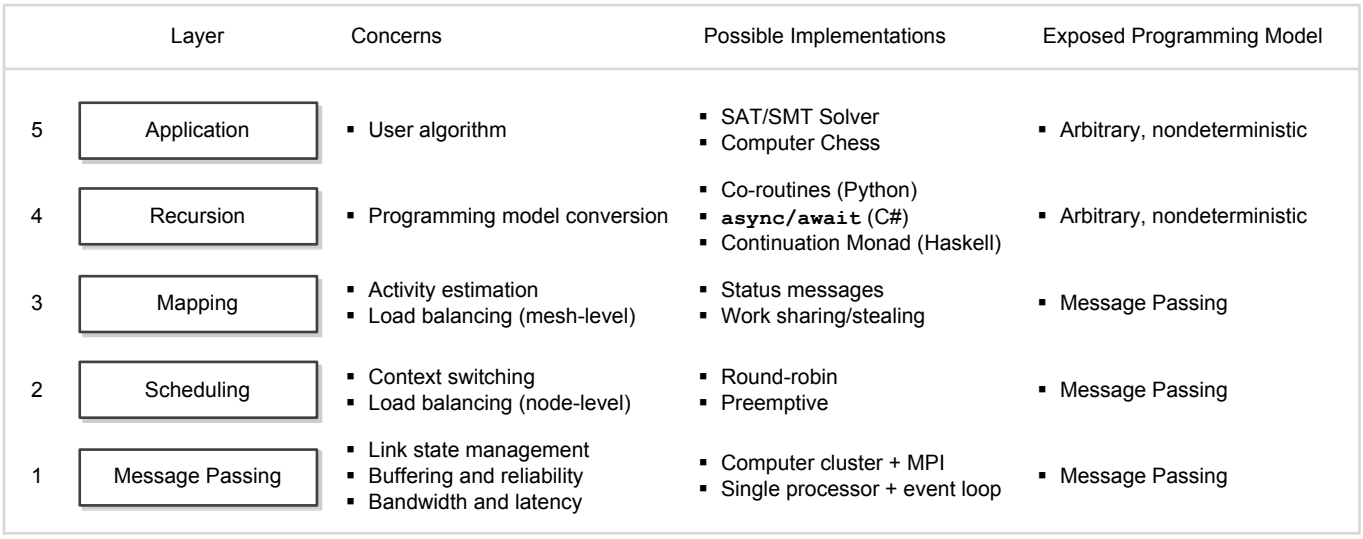


Figure 2. Proposed software model. Each layer addresses specific concerns when developing software for a hyperspace computer. The model decouples these concerns and makes it easier to tackle them independently. At the top layer (application), message passing and load balancing details are hidden from users and applications can be expressed in arbitrary high-level programming styles (e.g. imperative, functional).

predominantly based on message passing, usually through language extensions that provide direct access to message passing hardware facilities. Similarly, both the architecture and software of SpiNNaker are based on an asynchronous event-driven model. When a message arrives at a SpiNNaker core, it triggers an interrupt and causes execution to jump to a special handler. SpiNNaker programs are correspondingly expressed as initialization routines and event handlers [23], [24].

The direct correspondence between software and architecture means that all the machines discussed above assume *programming models* that are based on their underlying *execution models* [25]. This simplifies hardware/software co-design and makes hardware organization evident in software constructs [22]. These advantages, however, come with costs. First, programmers must be familiar with the architecture to develop efficient software that runs on top of it. Second, when an efficient program to solve a given problem is produced, the result is often tied to the specifics of the architecture and has poor portability [26]. Even across platforms with similar execution models, variations in hardware topology or supported features mean that programs must be adapted or entirely re-written. For example, when porting a message-passing program, changes in scale or topology may prompt non-trivial code changes [27]. Third, programs must contain intertwined portions of problem logic and low-level hardware calls, resulting in poor readability and maintainability [28].

The status quo in massively-parallel and hyperspace programming models is considerably different compared to desktop computing. Commodity platforms have deep stacks of abstraction layers that isolate users from architectural details and support a rich variety of programming paradigms (e.g. imperative, functional, dataflow and event-driven). Each of these paradigms is an ecosystem of software tools and design patterns that have proved useful in addressing different classes of computational problems [29]. Parallel and hyperspace machines on the other hand support fewer (and often unfamiliar) programming models that are directly based on their execution models. This lack of software support contributes to the lower appeal of these platforms and is recognized as one of the main challenges facing parallel computing in general [30]–[32].

III. Software Model

This section presents a multi-layer software abstraction model to develop combinatorial solvers (henceforth *solvers*) for hyperspace computers. In this context a *hyperspace computer* is a parallel machine consisting of a regular mesh of processors, typically representing a high-dimensional space. We assume that solvers work by "unfolding" the solution space across the mesh dynamically while searching for valid or optimal solutions.

The model provides a solver development framework to overcome the issues discussed in Section II-B, namely (1) abstracting away architectural details, (2) improving code reusability and maintainability and (3) providing end-users with high-level programming models. The key feature of the model is *hierarchical organization*: solver development is split into a number of concerns that can be addressed independently at different abstraction levels (Figure 2). We discuss the organization and advantages of the model below.

A. Layer Organization

1) Message Passing: The base layer consists of a computer architecture that can emulate a message passing system. This can be an architecture with bare-metal support for message passing (e.g. SpiNNaker), a network of interconnected

processors (commodity computers on an Ethernet network using MPI) or a software event loop running on a single processor. This layer handles data communication concerns and exposes a message passing interface while hiding hardware details from the layers above.

2) Scheduling: This layer maintains a number of concurrent processes that communicate via the message passing functions provided by layer 1. Each process has a *state* that is initialized at startup and then transformed by a handler function when a message is received. The layer is responsible for scheduling if processes are more numerous than hardware threads. Processes can be mapped to threads or managed at user level using microthreads or concurrency libraries [33], [34]. This layer allows top layers to run applications that are expressed as state initialization and message handling functions.

3) Mapping: The third layer is responsible for balancing work across the mesh. Similar to layer 2, it allows upper layers to run applications expressed as message handling routines. However, it prevents communication between arbitrary nodes and instead allows the application to request that a message be delivered without specifying its destination. The destination is then chosen based on estimated activity levels in subregions of the mesh. Since messages cannot be identified by their source or destination, alternative means of identifying related messages must be used. For example, applications can include identifying information in messages and quote them in later correspondence. Internally, this layer can rely on message queues and work stealing mechanisms to distribute messages.

4) Recursion: While bottom layers are responsible for providing a reliable, efficient and load-balanced message passing interface, the purpose of layer 4 is to hide message passing entirely and run recursive applications written in a high-level programming model (e.g. imperative or functional). The conversion between message passing and the target programming model is achieved using *continuation*: the ability to suspend a program, preserve its state then resume its execution sometime later. This feature is used as follows. First, the recursive function starts executing sequentially in a node A. When a recursive call is encountered, a continuation of the function is saved in a lookup table at A, alongside a unique callback identifier id. The recursive call parameters and callback identifier are then sent as a message to another node B which then repeats the same process. When B finishes executing, it returns the function result to A in a message, quoting id. Node A then looks up and invokes the continuation. This process is executed behind the scenes in a manner that is transparent to the user and application.

5) Application: At this layer of the model, message passing, scheduling, mapping and other issues have been abstracted away. Users now have a high-level programming model that allows them to focus on expressing problem logic and developing their applications. With few syntactic modifications, this layer can execute arbitrary recursive functions in the programming model provided by Layer 4.

B. Model Advantages

1) Separation of Concerns: The model decouples a number of issues that are encountered when developing applications for hyperspace computers. Loose coupling enables solutions to these issues to be implemented as independent modules which can then be integrated, exchanged, reused or revised without affecting each other. For example, if a more efficient mapping algorithm for a particular topology is developed (layer 3), it can be integrated into existing applications without affecting modules in other layers. This can be compared to a situation where, for example, an application is implemented as a "flat" message passing program. Changing the problem mapping procedure would then require changes throughout the program, possibly affecting other parts that were based on the old mapping implementation (e.g. message passing).

2) Problem-specific Tuning: In practice, it is unlikely that any single implementation at a given layer will be best for all applications. An application that makes a fixed number of recursive subcalls, for example, has a predictable unfolding behaviour and may be more efficiently executed by a static mapping algorithm. A static mapper does not exhaust the underlying message transfer infrastructure by exchanging status updates to maintain an accurate record of activity levels within the mesh. The modularity provided by the model makes it easier to explore such options and re-use good solutions to develop applications that share similar characteristics.

3) Cross-layer Optimization: One disadvantage of having strict decoupling between layers is that high level information is hidden from low-level algorithms. While this makes it easier to address problems independently (Section III-B1), it also prevents low-level algorithms from exploiting this information to improve performance. For example, solvers often employ lazy evaluation functions to prune the search space if the likelihood of finding a good solution falls below a certain margin. This heuristic can serve as an estimate of sub-problem size and hence the amount of computation likely to ensue. Mapping algorithms can exploit such knowledge to further optimize load balancing across the mesh (e.g. by delegating larger sub-problems to less utilized sub-regions of the mesh). The model allows such information to "fall through" by providing optional means to pass information to lower layers. This mechanism can be implemented via optional parameters that are passed at layer interfaces. At the application layer, end users can use software constructs such as annotations, decorators or pragmas to pass information to lower layers.

IV. Implementation

We present and discuss a prototype implementation of the proposed model, with emphasis on the mapping and recursion layers specifically (layers 3 and 4). We do not aim to present algorithms that are necessarily optimal (or even efficient). Instead, we focus on presenting a concrete implementation as an example of the organization outlined in Section III.

A. Message Passing and Scheduling (Layers 1 & 2)

We implemented a software backend to simulate a message passing system on a single processor. The backend initializes an array of node states and message queues then runs an event loop to deliver messages. On each simulation time step, a message is popped from each non-empty queue and passed to a handler function (`receive`) to update the respective node's state. While executing `receive`, the node can queue further messages for transmission using a send handler (`send`). The following is a simple application that can run on this backend:

```
1  function init(node):
2      state ← {visited: False}
3      return state
4
5  function receive(node, state, sender, msg, send
       , neighbours):
6      if state[visited] = False then
7          state[visited] ← True
8          foreach n in neighbours do
9              send(n, EMPTY_MSG)
```

Listing 1. Message-passing node traversal algorithm

In the above example, the initial state of each node is computed by an initialization function (`init`), in this case a dictionary comprising a Boolean field (`visited`). The receive handler implements a basic traversal algorithm: if a message arrives and `visited` is false then each neighbouring node is sent an empty message (`EMPTY_MSG`), where the list of neighbours (`neighbours`) is determined by topology. After initializing all nodes, the backend kickstarts computations by sending `EMPTY_MSG` to a user-selected node.

B. Mapping (Layer 3)

In the mapping layer we replace node identifiers with a *ticket system* that selects message destinations automatically. We introduce a slightly modified receive handler that replaces sender identity with a unique identifier (a *ticket*) that can be quoted to send reply messages. If no ticket is quoted, the send handler selects the destination automatically based on estimated activity levels within the node mesh. An example application that uses this programming style is shown below:

```
1  function receive(state, ticket, msg, send):
2      if msg = Call(n) then
3          if n<1 then
4              send(Result(0), ticket)
5          else
6              ticket ← send(Call(n−1))
7              state ← Continue(ticket, n)
8      else if msg = Result(total) then
9          if state = Continue(ticket, n) then
10             send(Result(total + n), ticket)
11         else
12             state ← Done(total)
13     else if msg = Trigger then
14         send(Call(10))
```

Listing 2. Message-passing algorithm to calculate the sum 1 to 10

This application is message-passing implementation of the recursive function

$$\text{sum}(n) = \begin{cases} 0, & n < 1 \\ n + \text{sum}(n-1), & \text{otherwise} \end{cases}$$

and is here used to calculate sum(10), as follows. An incoming message is first classified as an (1) evaluation call, (2) a returned result or (3) an initialization trigger. Evaluation calls are handled in lines 2–7: if $n < 1$ then a reply message with payload `Result(0)` is sent immediately, quoting the incoming message's ticket number (base case). Otherwise, a subcall message is sent requesting another node to evaluate $\text{sum}(n-1)$, and the parent call ticket is stored for bookkeeping (alongside n). When the result is returned (lines 8–10), it is added to n then returned to the parent node, quoting the stored ticket number.

To start the computation, the backend sends a trigger message to a user-selected node K, causing it to make a subcall to evaluate sum(10) (line 14). This initiates a chain of subcalls, spanning multiple nodes, which then terminates with a back propagation of result messages. When K finally receives a result message, it stores the value $\text{sum}(10)$ in a field `total` that is read by the user (line 12).

C. Recursion (Layer 4)

The programming pattern used in Listing 2 provides a way to implement generic recursive functions as message passing algorithms. However, its control flow is difficult to trace and likely to become unwieldy for anything but trivial recursive functions (especially if more features were introduced, e.g. concurrent subcalls). We resolve this by implementing an extended form of this programming pattern independently as layer 4. This layer allows users to express their algorithms in a more concise form, converting subcalls to messages behind the scenes and hiding the complexity of call bookkeeping.

Figure 3 shows a basic mechanism to implement layer 4 using Cilk-like syntax [35]. The purpose of this mechanism is to (1) implement a form of fork-join parallelism and (2) use the underlying message passing system in layer 3 to delegate subcalls to other nodes. Here, we wish to "intercept" recursive subcalls, suspend the current context then resume execution when the subcall result is returned. Although this can be implemented using multi-threading, many modern programming languages support lightweight forms of user-managed threads that are better suited for our purposes. Here, we use a `yield` operator as a mechanism for communication between layer 4 and application code. `yield` is a modified return statement that returns execution to the parent function while preserving the current context. The parent function (layer 4) may then resume executing the context later, and optionally pass data to the application as the result of evaluating the `yield` statement.

The proposed mechanism (Figure 3) works as follows. Layer 4 maintains a record of invoked calls (call records). When the recursive function wants to make a subcall, it yields

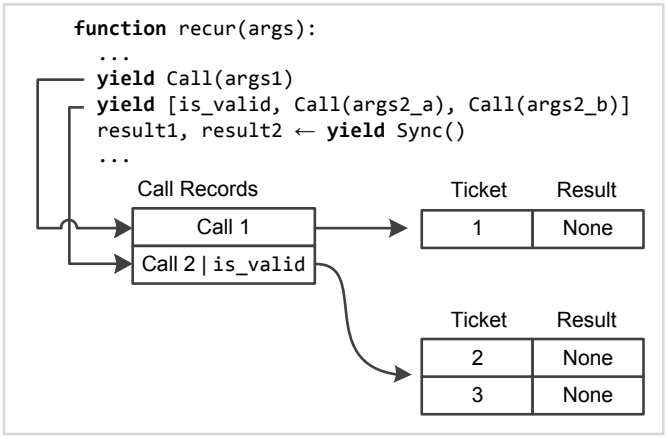


Figure 3. An implementation of layer 4 using the `yield` operator

a `Call` object with the subcall parameters (first yield statement in Figure 3). Layer 4 then sends a message through layer 3, asking another node to evaluate the subcall. The ticket number issued by layer 3 is stored in call records, alongside an empty slot for a pending computation result. Anytime layer 3 returns a result message, its ticket number is inspected and the payload (evaluation result) is stored in the result field of the appropriate call record. The recursive function can receive the results of the last calls made by yielding a `Sync` object. Layer 4 will then lookup and return the results of all recent subcalls from call records (blocking execution if any results are still pending).

This pattern can be extended to support non-deterministic choice as follows. The recursive function yields a list consisting of a validation function is_valid and several `Call` objects (second yield statement in Figure 3). Layer 4 issues multiple messages, corresponding to each subcall, then stores all tickets in the same call record. When a subcall evaluation e is received and $is\_valid(e) = \text{true}$, execution resumes (returning e as a result) and the remaining evaluations are ignored. If all evaluations are returned but none satisfy is_valid then a $null$ value is returned to the application.

D. Application (layer 5)

The advantage of extracting then hiding subcall message handling logic become obvious when considering the re-implementation of Listing 2 in the style provided by layer 4:

```
1  function calculate_sum(n):
2      if n<1 then
3          yield Result(0)
4      else
5          yield Call(n−1)
6          total ← yield Sync()
7          yield Result(total + n)
```

Listing 3. An algorithm to calculate the sum 1 to N recursively

This implementation contains application logic only and is therefore more concise, readable and easy to maintain. Behind the scenes, layers 1 through 4 continue to manage message delivery, scheduling and load balancing.

V. Evaluation

We describe the implementation and optimization of a Boolean satisfiability (SAT) solver as a use case for the proposed model. Due to the theoretical and growing practical relevance of SAT, considerable effort continues to be invested in developing ever more powerful SAT solvers, particularly using parallel approaches [36]–[38]. However, parallelizing SAT is hampered by the difficulty of workload balancing on shared memory architectures [39], [40]. This makes the problem a good candidate for solving on a hyperspace computer and a practical use case for evaluating the model.

A. Simulated Architectures

We used the simulation backend described in Section IV-A for our evaluation. The machines we simulated had hyper-torus topologies of either 2 or 3 dimensions and varied in number of cores. We assumed that messages can be communicated between adjacent cores only (no underlying message delivery network) and that inter-node message queues were sufficiently large to accommodate all pushed messages without blocking send handlers. As a baseline for comparison, we also simulated fully-connected machines under the same assumptions.

B. Solver Implementation

The solver is based on a barebone implementation of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm (Listing 4), working as follows. First, the problem is simplified using procedures that can infer variable assignments, either because a variable is the only unassigned one in a clause (unit propagation, lines 6–8) or because it always occurs with the same polarity (pure literal assignment, lines 9–11). The simplified problem is then decomposed into two sub-problems by selecting one variable (using an algorithm-independent heuristic) and assigning it true and false values (lines 12–14). Finally, the subproblems are evaluated concurrently using the mechanism for non-deterministic choice outlined in Section IV-C (line 15). This ensures that, if a solution to one of the sub-problems is found, the application will resume execution without waiting for other result.

```
1  function solve_sat(problem):
2      if consistent(problem) then
3          yield Result(SAT)
4      if exist_empty_clause(problem) then
5          yield Result(UNSAT)
6      for clause in problem[clauses] do
7          if unit_clause(clause) then
8              unit_propagate(problem, clause)
9      for literal in problem[literals] do
10         if literal_pure(literal) then
11             assign_pure(problem, literal)
12     L ← select_literal(problem)
13     subp1 ← assign(problem, L, True)
14     subp2 ← assign(problem, L, False)
15     yield [is_SAT, Call(subp1), Call(subp2)]
16     result ← yield Sync()
17     yield result
```

Listing 4. DPLL algorithm for solving Boolean satisfiability problems

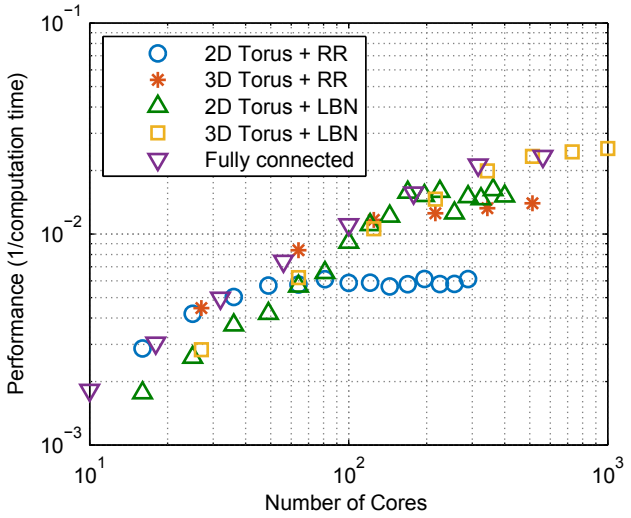


Figure 4. Comparison of SAT solver scalability for different topologies and mapping algorithms. RR is round-robin mapping and LBN is least-busy-neighbour mapping. Each data point is the average performance over 20 benchmark SAT problems from [42].

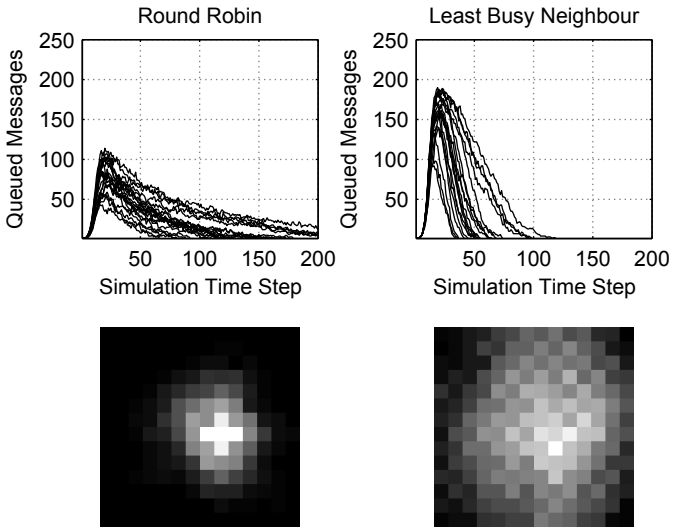


Figure 5. The temporal and spatial unfolding of SAT problems for the two mapping algorithms. Top row plots are superimposed simulation traces for different problems running on a 196-core 2D torus machine. Bottom row plots are heatmaps of total messages delivered across the mesh for one problem.

In practice, many state-of-the-art SAT solvers implement additional heuristics such as *conflict-driven learning* and *non-chronological backtracking* to prune the search space [41]. However, our focus here is (1) finding efficient ways to map problem instances across the mesh and (2) evaluating the impact of topology on scalability. To this end we choose a basic implementation of DPLL for our evaluation.

C. Simulation Procedure

Because the application code for the SAT solver (Listing 4) is decoupled from the mapping algorithm, we can profile and optimize the application's mapping process independently. To do so, we chose a collection of uniform random 3-SAT problems for our benchmark (20 variables and 91 clauses each, all satisfiable [42]). For each simulation run, we configured the simulator backend to output the state of each message queue buffer as a time series, covering the entire simulation. From the output log we calculated:

(1) *Computation time*: the number of simulation time steps between the first (trigger) and last messages,

(2) *Interconnect activity*: the total number of queued messages across the mesh versus time, and

(3) *Node activity*: the total messages delivered to each node during the simulation.

We use computation time to compare performance across runs and interconnect/node activities to profile the temporal and spatial unfolding of computations across the mesh.

D. Mapping, Topology and Dimensionality

We ran a series of simulations to explore the impact of mapping algorithm, hyperspace topology and dimensionality on the absolute performance and scalability of our SAT solver application. We classify mapping algorithms as *static* or *adaptive* depending on whether their behaviour is determined apriori or influenced by the runtime behaviour of the application. Adaptive mapping algorithms involve under-the-hood message-passing mechanisms to estimate activity levels between neighbouring cores and map problem instances correspondingly. We chose to compare two mapping algorithms, one of each class:

(1) *Round robin (static)*: map sub-problems to adjacent cores in circular order.

(2) *Least busy neighbour (adaptive)*: Embed a count of total messages received in all outgoing messages and maintain a record of neighbouring node counts. Map sub-problems to neighbour with the smallest count.

Figure 4 shows scalability plots for the different topologies and mapping algorithms. We observed that increasing dimensionality and using adaptive (least-busy-neighbour) mapping improved solver scalability. Adaptive mapping had a negative impact on absolute performance for smaller topologies (<100 cores) but improved performance significantly at larger sizes. The benefit of adaptive mapping is almost as pronounced as increasing dimensionality; large 2D machines with adaptive mapping performed just as well as 3D machines with static (round-robin) mapping while large 3D machines with adaptive mapping performed nearly like fully connected machines.

E. Spatial and Temporal Unfolding

Figure 5 shows interconnect and node activity plots obtained by simulating the SAT solver on a 196-core 2D torus. These results are consistent with the performance trends observed in Figure 4. Least-busy-neighbour mapping results in a larger degree of spatial unfolding, more astute message queuing and hence faster execution compared to round-robin mapping.

VI. Discussion

We discuss the relationship between our model and some work in literature. In particular, we compare it to (1) existing software abstractions that build on top of message passing and (2) parallel programming extensions for high-level languages.

A. Related Models and Abstractions

Many message passing platforms either support or are built from the grounds up to conform to the Message Passing Interface (MPI) standard. MPI provides a common base for developing portable message-passing applications that execute efficiently on different architectures. Even though the standard hides hardware details from users, developing MPI programs involves controlling data and control flows at a fine level of detail and can therefore be viewed as a form of low-level programming [43], [44]. The prevalence of message passing in parallel computing is not surprising since (1) low-level programming allows fine-grained performance optimizations and (2) parallel applications are developed specifically for high performance. Low-level programming, however, compromises code expressiveness and user productivity, both important factors in the software development cycle [45]. Several parallel programming models and libraries that build on message passing attempt to address this by either:

(1) Implementing common functionality likely to be needed by many parallel applications (e.g. fault-tolerance [46]),
(2) Addressing specific shortcomings and development needs (e.g. reusability [47] and verifiability [48]), or
(3) Providing complete solutions for specific applications (e.g. neural simulation using NEST [49]).

In general, these abstractions tend to gravitate towards being either too thin (1 and 2) or too deep (3), catering to either performance or expressiveness. The proposed hierarchical model combines the best of both worlds; it provides a high-level programming interface at the top and yet supports a high degree of tuning to meet performance goals.

B. Related Languages and Extensions

Several language extensions and independent programming languages provide design patterns and constructs for parallel programming in styles other than message passing. For example, OpenMP adds work sharing constructs to C [50] and Cilk similarly provides fork-join semantics [35]. These extensions are similar to the proposed model in that they expose a high-level programming style to isolate users from parallel execution details. However, there are some notable differences. Both OpenMP and Cilk target shared memory architectures and are concerned primarily with handling scheduling and multi-threading on behalf of users. Hyperspace architectures are distributed systems where the individual processors are not fully connected and therefore cannot be managed by a global scheduler. In addition, due to the massive scale of hyperspace systems, additional concerns that are not handled by these frameworks such as inter-node communication and problem decomposition/mapping play a central role in maximizing the degree of parallelism. Therefore, these frameworks are not independently suitable for hyperspace computers. Nevertheless, they can be integrated into the proposed model as implementations for layer 2.

C. Generality and Limitations

We have discussed the proposed model as a software framework to develop combinatorial solvers that are expressed as single recursive functions. However, the fork-join mechanism exposed at the top layer is in fact more general. The model can therefore be used to develop other types of applications, including ones with multiple functions and complex call graphs. Even though this is within the model's remit, such applications have problem domains that are difficult to describe and map, making them less attractive for porting to hyperspace computers [13].

The question also arises of whether scientific computing problems can be expressed using the proposed model. As is the case with desktop computing, no single programming paradigm is likely best to encode and compute all types of problems. Many scientific problems involve exchanging state updates between nodes and this may best be encoded using message passing. In addition, scientific problems often have a static problem domain that is known apriori and does not require dynamic/adaptive mapping to the core mesh. In this case, the functional equivalent of layers 1 and 2 (message passing and scheduling) may be sufficient to encode and compute scientific problems efficiently.

VII. Conclusion

We presented a hierarchical software abstraction model to develop combinatorial solvers for massively-parallel machines with regular topologies. The model decouples several concerns in the design of these solvers including inter-node communication, node-level scheduling, problem mapping, mesh-level load balancing and expressing problem logic. In addition, it can expose arbitrary programming models on top of the underlying message passing system, enabling users to re-use existing design patterns, libraries and algorithms in high-level programming styles. The proposed hierarchy allows users of massively parallel machines to combine the expressiveness of high-level programming with the fine degree of performance tuning permissible by low-level programming. One possible realization of the model is to have a repertoire of modules (representing alternative implementations for each layer) tuned for specific architectures or applications. This repertoire can be populated by both architecture and application developers. New applications for hyperspace machines can then be developed quickly by assembling the appropriate set of modules from this repertoire and configuring or modifying them to suit the application as needed.

Acknowledgments

This work was supported by EPSRC grant EP/N031768/1 (project POETS).

REFERENCES

[1] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao *et al.*, "The Sunway TaihuLight supercomputer: system and applications," *Science China Information Sciences*, vol. 59, no. 7, p. 072001, 2016.

[2] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The SpiNNaker Project," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 652–665, May 2014. [Online]. Available: http://dx.doi.org/10.1109/jproc.2014.2304638

[3] G. C. Fox, "What have we learnt from using real parallel machines to solve real problems?" in *Proceedings of the third conference on Hypercube concurrent computers and applications-Volume 2*. ACM, 1989, pp. 897–955.

[4] R. Martins, V. Manquinho, and I. Lynce, "An overview of parallel SAT solving," *Constraints*, vol. 17, no. 3, pp. 304–347, 2012.

[5] J. P. Hayes, T. N. Mudge, Q. F. Stout, S. Colley, and J. Palmer, "Architecture of a Hypercube Supercomputer." in *ICPP*, 1986, pp. 653–660.

[6] C. Whitby-Strevens, "The transputer," in *ACM SIGARCH Computer Architecture News*, vol. 13, no. 3. IEEE Computer Society Press, 1985, pp. 292–300.

[7] A. J. Hey, "Transputers, OCCAM and general-purpose parallel computing," *Proceedings: 1989 CERN School of Computing, Bad Herrenalb, Federal Republic of Germany, 20 August-2 September 1989*, vol. 90, no. 6, p. 93, 1990.

[8] C. A. R. Hoare, "Communicating sequential processes," in *The origin of concurrent programming*. Springer, 1978, pp. 413–443.

[9] K. R. Apt, N. Francez, and W. P. De Roever, "A proof system for communicating sequential processes," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 2, no. 3, pp. 359–385, 1980.

[10] G. M. Reed and A. W. Roscoe, "A timed model for communicating sequential processes," *Theoretical Computer Science*, vol. 58, no. 1, pp. 249–261, 1988.

[11] D. J. Pritchard, C. Askew, D. Carpenter, I. Glendinning, A. J. Hey, and D. A. Nicole, "Practical parallelism using transputer arrays," in *International Conference on Parallel Architectures and Languages Europe*. Springer, 1987, pp. 278–294.

[12] J. P. Hayes, T. Mudge, Q. F. Stout, S. Colley, and J. Palmer, "A microprocessor-based hypercube supercomputer," *IEEE micro*, vol. 6, no. 5, pp. 6–17, 1986.

[13] J. Hayes and T. Mudge, "Hypercube supercomputers," *Proceedings of the IEEE*, vol. 77, no. 12, pp. 1829–1841, 1989. [Online]. Available: http://dx.doi.org/10.1109/5.48826

[14] M.-Y. Chan, "Embedding of d-dimensional grids into optimal hypercubes," in *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*. ACM, 1989, pp. 52–57.

[15] S. N. Bhatt and I. C. Ipsen, "How to Embed Trees in Hypercubes." DTIC Document, Tech. Rep., 1985.

[16] K. Efe, "Embedding mesh of trees in the hypercube," *Journal of Parallel and Distributed Computing*, vol. 11, no. 3, pp. 222–230, 1991.

[17] M. T. Heath *et al.*, *Hypercube Multiprocessors 1986*. Siam, 1986.

[18] ——, *Hypercube Multiprocessors, 1987: Proceedings of the Second Conference on Hypercube Multiprocessors, Knoxville, Tennessee, September 29-October 1, 1986*. Siam, 1987, vol. 29.

[19] A. D. Brown, R. Mills, K. J. Dugan, J. S. Reeve, and S. B. Furber, "Reliable computation with unreliable computers," *IET Computers & Digital Techniques*, vol. 9, no. 4, pp. 230–237, 2015.

[20] P. Walker, "Transputer." *Byte*, vol. 10, no. 5, pp. 219–237, 1985.

[21] N. H. Gehani and W. D. Roome, "Rendezvous facilities: Concurrent C and the Ada language," *IEEE Transactions on Software Engineering*, vol. 14, no. 11, pp. 1546–1553, 1988.

[22] R. Dettmer, "Occam and the transputer," *Electronics and Power*, vol. 31, no. 4, pp. 283–287, 1985.

[23] A. D. Rast, X. Jin, F. Galluppi, L. A. Plana, C. Patterson, and S. Furber, "Scalable event-driven native parallel processing: the SpiNNaker neuromimetic system," in *Proceedings of the 7th ACM international conference on Computing frontiers*. ACM, 2010, pp. 21–30.

[24] A. D. Brown, S. B. Furber, J. S. Reeve, J. D. Garside, K. J. Dugan, L. A. Plana, and S. Temple, "SpiNNaker - Programming Model," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1769–1782, 2015.

[25] L. Bouge, "The data parallel programming model: A semantic perspective," in *The Data Parallel Programming Model*. Springer, 1996.

[26] J. D. Mooney, "Portability and reusability: common issues and differences," in *Proceedings of the 1995 ACM 23rd annual conference on Computer science*. ACM, 1995, pp. 150–156.

[27] J. Y. Cotronis, "Reusable message passing components," in *Parallel and Distributed Processing, 2000. Proceedings. 8th Euromicro Workshop on*. IEEE, 2000, pp. 398–405.

[28] J. R. Donaldson, "Structured programming," in *Classics in software engineering*. Yourdon Press, 1979, pp. 179–185.

[29] P. Van Roy *et al.*, "Programming paradigms for dummies: What every programmer should know," *New computational paradigms for computer music*, vol. 104, 2009.

[30] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek *et al.*, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[31] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[32] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2016.

[33] C. Tismer, "Stackless python," 2000.

[34] J. M. Bjørndalen, B. Vinter, and O. J. Anshus, "PyCSP-Communicating Sequential Processes for Python." in *CPA*, 2007, pp. 229–248.

[35] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*. ACM, 1995, vol. 30, no. 8.

[36] W. Chrabakh and R. Wolski, "GrADSAT: A parallel sat solver for the grid," in *Proceedings of IEEE SC03*, 2003.

[37] Y. Hamadi, S. Jabbour, and L. Sais, "ManySAT: a parallel SAT solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 245–262, 2008.

[38] C. Sinz, W. Blochinger, and W. Küchlin, "PaSATâĂŤParallel SAT-checking with lemma exchange: Implementation and applications," *Electronic Notes in Discrete Mathematics*, vol. 9, pp. 205–216, 2001.

[39] M. Böhm and E. Speckenmeyer, "A fast parallel SAT-solverâĂŤEfficient workload balancing," *Annals of Mathematics and Artificial Intelligence*, vol. 17, no. 2, pp. 381–400, 1996.

[40] H. Zhang, M. P. Bonacina, and J. Hsiang, "PSATO: a distributed propositional prover and its application to quasigroup problems," *Journal of Symbolic Computation*, vol. 21, no. 4, pp. 543–560, 1996.

[41] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, 1999.

[42] "Satlib - benchmark problems," http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html, accessed: 2017-03-28.

[43] C. Rodrigues, "Supporting high-level, high-performance parallel programming with library-driven optimization," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2014.

[44] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, L. Snyder, W. D. Weathersby, and C. Lin, "The case for high-level parallel programming in ZPL," *Ieee computational science and engineering*, vol. 5, no. 3, pp. 76–86, 1998.

[45] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

[46] J. J. Wilke, K. Teranishi, J. C. Bennett, H. Kolla, D. S. Hollman, and N. Slattengren, "Evolving the Message Passing Programming Model via a Fault-Tolerant, Object-oriented Transport Layer," in *Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale*. ACM, 2015, pp. 41–46.

[47] J. Cotronis, "Message-passing program development by ensemble," *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 242–249, 1997.

[48] J. Carter and W. B. Gardner, "A formal CSP framework for message-passing HPC programming," in *Electrical and Computer Engineering, 2006. CCECE'06. Canadian Conference on*. IEEE, 2006, pp. 1466–1470.

[49] M.-O. Gewaltig and M. Diesmann, "Nest (neural simulation tool)," *Scholarpedia*, vol. 2, no. 4, p. 1430, 2007.

[50] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.